Case study: FIXML to Java, C# and C++

K. Lano, S. Yassipour-Tehrani, K. Maroukian
Dept. of Informatics, King's College London, Strand, London, UK

This case study is a transformation from financial transaction data expressed in FIXML XML format, into class definitions in Java, C# and C++. It is based on an industrial application of MDD in finance, and aims to support rapid upgrading of user software when new or extended FIXML definitions become available. The transformation involves text-to-model, model-to-model and model-to-text subtransformations.

1 Introduction

Financial transactions can be electronically expressed using formats such as the FIX (Financial Information eXchange) format. New custom variants/extensions of such message formats can be introduced, which leads to problems in the maintainance of end-user software: the user software, written in various programming languages, which generates and processes financial transaction messages will need to be updated to the latest version of the format each time it changes. In [2] the authors proposed to address this problem by automatically synthesising program code representing the transaction messages from a single XML definition of the message format, so that users would always have the latest code definitions available. For this case study we will restrict attention to generating Java, C# and C++ class declarations from messages in FIXML 4.4 format, as defined at http://fixwiki.org/fixwiki/FPL:FIXML_Syntax, and http://www.fixtradingcommunity.org.

The solution transformation should take as input a text file of a message in XML FIXML 4.4 Schema format, and produce as output corresponding Java, C# and C++ text files representing this data.

2 Core problem

The solution transformation should be broken down into the following subtransformations:

1. XML text to model of XML metamodel (Figure 1)
2. model of XML metamodel to a model of a suitable metamodel for the programming language/languages under consideration
3. program model to program text.

By using a chain of transformations, greater flexibility and extensibility is supported: language mapping issues at the abstract syntax level can be separated from concrete syntax mapping, and generation of text in an additional programming language may involve only the definition of a new model to text transformation, and possibly the definition of a new/enhanced programming language metamodel and model-to-model transformation. The XML text to model transformation does not need to change. We have found that a single programming language metamodel and model-to-model transformation is sufficient for Java, C# and C++.

To appear in EPTCS.

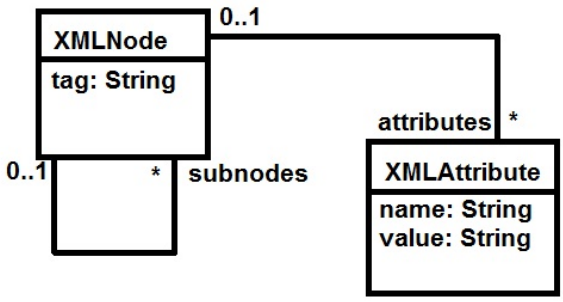

Figure 1: XML metamodel

Solutions to the case study can devise their own metamodel(s) for the abstract syntax of the target programming languages. Solutions may use external software for the XML parsing step and/or for the code generation step, and may use different transformation languages for the 3 subtransformations.

The informal transformation rules mapping from XML to a programming language (eg., Java) are the following, in terms of concrete syntax:

- (Rule 1): An XML tag is translated to a Java Class:

  ```
  <tag1 .... />
  ```

  becomes

  ```
  class tag1 { .... }
  ```

- (Rule 2): XML attributes are mapped to Java attributes:

  ```
  <tag1 att1="val1" att2="val2" />
  ```

  becomes

  ```
  class tag1
  { String att1 = "val1"; String att2 = "val2"; }
  ```

  etc.

- (Rule 3): Nested XML tags become Java member objects:

  ```
  <tag1 .... >
    <tag2 ... />
    <tag3 ... />
  </tag1>
  ```

  becomes

  ```
  class tag1
  { ....
    tag2 tag2_object = new tag2();
    tag3 tag3_object = new tag3();
  }
  ```

This rule should also take that case into account where multiple subnodes of the same node with the same tag name exist: these subnodes may be represented by distinct attributes with the same tag object type, initialised by specific constructors, or by an array/list of such objects.

In order to improve the utility of the generated program code, constructors should be provided for the generated classes, which permit initialising of all their features. A default no-argument constructor should also be provided.

As an example, the XML data

```
<car make="XJ6" colour="silver" manufacturer="Jaguar">
  <engine capacity="3l" />
  <DVLARecord status="OffRoad" date="1.1.12"/>
</car>
```

becomes in Java:

```
class engine
{ String capacity = "3l";

  engine() {}

  engine(String capacity)
  { this.capacity = capacity; }
}

class DVLARecord
{ String status = "OffRoad";
  String date = "1.1.12";

  DVLARecord() {}

  DVLARecord(String status, String date)
  { this.status = status; this.date = date; }
}

class car
{ String make = "XJ6";
  String colour = "silver";
  String manufacturer = "Jaguar";
  engine engine_object = new engine();
  DVLARecord DVLARecord_object = new DVLARecord();

  car() {}

  car(String make, String colour, String manufacturer,
        engine engine_, DVLARecord DVLARecord_)
  { this.make = make; this.colour = colour;
    this.manufacturer = manufacturer;
    this.engine_object = engine_;
```

```
    this.DVLARecord_object = DVLARecord_;
  }
}
```

In C# it would be:

```
class engine
{ string capacity = "3l";

  engine() {}

  engine(string capacity)
  { this.capacity = capacity; }
}

class DVLARecord
{ string status = "OffRoad";
  string date = "1.1.12";

  DVLARecord() {}

  DVLARecord(string status, string date)
  { this.status = status; this.date = date; }
}

class car
{ string make = "XJ6";
  string colour = "silver";
  string manufacturer = "Jaguar";
  engine engine_object = new engine();
  DVLARecord DVLARecord_object = new DVLARecord();

  car() {}

  car(string make, string colour, string manufacturer,
        engine engine_, DVLARecord DVLARecord_)
  { this.make = make; this.colour = colour;
    this.manufacturer = manufacturer;
    this.engine_object = engine_;
    this.DVLARecord_object = DVLARecord_;
  }
}
```

A C++ version could be:

```
class engine
{ private:
  string capacity;

  public:
  engine()
  { capacity = "3l"; }

  engine(string capacity_)
```

```
  { capacity = capacity_; }
};

class DVLARecord
{ private:
   string status;
   string date;

  public:
   DVLARecord()
   { status = "OffRoad";
     date = "1.1.12";
   }

   DVLARecord(string status_, string date_)
   { status = status_; date = date_; }
};

class car
{ private:
   string make;
   string colour;
   string manufacturer;
   engine* engine_object;
   DVLARecord* DVLARecord_object;

  public:
  car()
  { make = "XJ6";
    colour = "silver";
    manufacturer = "Jaguar";
  }

  car(string make_, string colour_, string manufacturer_,
        engine* engine_, DVLARecord* DVLARecord_)
  { make = make_; colour = colour_;
    manufacturer = manufacturer_;
    engine_object = engine_;
    DVLARecord_object = DVLARecord_;
  }
};
```

For C++, the class declarations should be ordered so that classes are always declared before they are used.

2.1 Test cases

The solutions should be tested on the test cases `test1.xml` to `test8.xml` provided. Test cases 1 to 4 represent typical FIXML messages. Tests 5 and 6 are tests of solution efficiency on large messages. Tests 7 and 8 are examples of invalid XML files which should be rejected by the transformation.

The first test is a simple example of an Order message:

```
<?xml version="1.0" encoding="ASCII"?>
<FIXML>
     <Order ClOrdID="123456"
                    Side="2"
                    TransactTm="2001-09-11T09:30:47-05:00"
                    OrdTyp="2"
                    Px="93.25"
                    Acct="26522154">
         <Hdr Snt="2001-09-11T09:30:47-05:00"
                 PosDup="N"
                 PosRsnd="N"
                 SeqNum="521">
             <Sndr ID="AFUNDMGR"/>
             <Tgt ID="ABROKER"/>
         </Hdr>
         <Instrmt Sym="IBM"
                    ID="459200101"
                    IDSrc="1"/>
         <OrdQty Qty="1000"/>
       </Order>
</FIXML>
```

The second test is a more complex example of a Position Report message, which features multiple subnodes with the same tagname:

```
<?xml version="1.0" encoding="ASCII"?>
<FIXML>
<PosRpt RptID="541386431" Rslt="0"
           BizDt="2003-09-10T00:00:00" Acct="1" AcctTyp="1"
           SetPx="0.00" SetPxTyp="1" PriSetPx="0.00" ReqTyp="0" Ccy="USD">
       <Hdr Snt="2001-12-17T09:30:47-05:00" PosDup="N" PosRsnd="N" SeqNum="1002">
<Sndr ID="String" Sub="String" Loc="String"/>
<Tgt ID="String" Sub="String" Loc="String"/>
<OnBhlfOf ID="String" Sub="String" Loc="String"/>
<DlvrTo ID="String" Sub="String" Loc="String"/>
</Hdr>
<Pty ID="OCC" R="21"/>
<Pty ID="99999" R="4"/>
<Pty ID="C" R="38">
<Sub ID="ZZZ" Typ="2"/>
</Pty>
<Qty Typ="SOD" Long="35" Short="0"/>
<Qty Typ="FIN" Long="20" Short="10"/>
<Qty Typ="IAS" Long="10"/>
<Amt Typ="FMTM" Amt="0.00"/>
<Instrmt Sym="AOL" ID="KW" IDSrc="J" CFI="OCASPS" MMY="20031122"
                     Mat="2003-11-22T00:00:00" Strk="47.50" StrkCcy="USD" Mult="100"/>
</PosRpt>
</FIXML>
```

If there are multiple nodes with the same tag, the class representing the tag will have the union of the instance variables derived from the attributes and subnodes of all these occurrences. For example, Qty is represented by

```
class Qty
{ String Type = "SOD";
  String Long = "35";
  String Short = "0";
  ...
}
```

Other sample FIXML messages can be found at http://fixwiki.org/fixwiki/FPL:FIXML_Syntax.

3 Extensions

The following enhancements of the transformation can be considered.

3.1 Selection of appropriate data types

In cases where attribute values are integers or doubles, the attributes should be mapped to programming language instance variables of these types. For example, `Strk="47.50"` would be mapped to `double Strk = 47.50;`.

3.2 Extension to additional languages

Identify how the transformation can be extended to generate C code instead of object-oriented language code.

3.3 Generic transformation

The transformation could also be extended to define a generic transformation which maps the FIXMLSchema definition (http://fixwiki.org/fixwiki/FPL:FIXML_Syntax) or DTD (http://www.fixtradingcommunity.org/pg/structure/tech-specs/fix-version/44) into Java, C# and C++. This mapping would support the comprehensive representation of arbitrary valid FIXML messages as program objects.

References

[1] Botella, P., Burgués, X., Carvallo, J. P., Franch, X., Grau, G., Marco, J., Quer, C., *ISO/IEC 9126 in practice: what do we need to know?*, Software Measurement European Forum (SMEF 2004).

[2] M. B. Nakicenovic, *An Agile Driven Architecture Modernization to a Model-Driven Development Solution*, International Journal on Advances in Software, vol 5, nos. 3, 4, 2012, pp. 308–322.

A Evaluation criteria

As the basis of a systematic evaluation framework for model transformations, we propose to use the International Organisation for Standardization (ISO) standards related to software quality, specifically the ISO/IEC 9126-1 standard, which is based upon the definition of a *Quality Model* and its use for software evaluation [1]. This framework defines quality models based on general characteristics of software, which are further refined into subcharacteristics. Relevant characteristics and subcharacteristics for evaluation of model transformations can be selected from the ISO/IEC 9126-1 framework. These characteristics and subcharacteristics can then be further decomposed into

Characteristic	Subcharacteristic	Attribute
Functionality	Suitability	Abstraction level Complexity Development effort Execution time
	Accuracy	Syntactic correctness Semantic preservation
Reliability	Fault tolerance	Detection/processing of invalid models
Maintainability	Changeability	Complexity Modularity

Table 1: Selected quality characteristics for the evaluation of model transformation approaches

measurable attributes. Table 1 summarizes the chosen characteristics, subcharacteristics and their corresponding measurable attributes. One attribute may be related to more than one quality factor.

The following are the specific measures which should be evaluated for each solution to this case study:

- Complexity: sum of number of operator occurrences and feature and entity type name references in the specification expressions
- Accuracy: that the resulting programs are valid in their languages (syntactic correctness), and that they correctly represent the source XML data structure and elements (semantic preservation). In particular, the programming language constraint that distinct instance variables of the same class must have distinct names must be ensured (syntactic correctness).
- Development effort: developer time in person-hours spent in writing and debugging the specification
- Fault tolerance: High if transformation is able to detect invalid input XML and produce accurate error messages; Medium if erroneous files produce a failed execution with an indication that some error occurred; Low if such files are processed and output produced without warnings being issued
- Execution time: milliseconds for execution of each of the three stages
- Modularity: $1 - \frac{d}{r}$ where d is the number of dependencies between rules (implicit or explicit calls, ordering dependencies, inheritance or other forms of control or data dependence) and r is the number of rules.

Abstraction level is classified as High for primarily declarative solutions, Medium for declarative-imperative solutions, and Low for primarily imperative solutions.

Execution time of the subtransformation implementations includes the loading of models and printing of output code from the transformation tool(s).